Popular Computing

Volume 8
Number 5
May 1980

86

Thomas R. Parkin:

Measuring Time: Part II

A Star Is Formed

Problem Solution

Problem 66 in issue 19 was the following:

A succession of random numbers (uniformly distributed in the range from 001 to 999) is drawn. The numbers are progressively totalled until the sum is a prime number, at which time the game ends and the score is the number of numbers drawn.

What is the distribution of the scores?

Bernard Kasten, of Kansas State University, wrote a program (in Microsoft BASIC) to attack this problem in a straightforward manner by generating random numbers and carrying out the indicated procedure. His result for 1700 plays is as follows:

Game length	number of games	Game length	number of games	Game length	number of games
1	577	8	24	15	3
2	351	9	0	16	4
3	269	10	15	17	0
4	171	11	15	18	1
5	117	12	10	19	1
6	78	13	7	20	2
7	48	14	7		

Publisher: **Audrey Gruenberger**
Editor: **Fred Gruenberger**
Associate Editors: **David Babcock, Irwin Greenwald, Patrick Hall**
Contributing Editors: **Richard Andree, William C. McGee, Thomas R. Parkin, Edward Ryan**
Art Director: **John G. Scott**
Business Manager: **Ben Moore**

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year. For all other countries, add $3.50 per year. Back issues $2.50 each. Copyright 1980 by POPULAR COMPUTING.

@ 2023 This work is licensed under CC BY-NC-SA 4.0

Measuring Time: Part II

by Thomas R. Parkin

Time is a relative phenomenon. The hiatus between Part I of this essay and this, Part II, can be characterized as relatively nil or inordinately long, depending on your point of view. The editor will incline to the latter; astronomically speaking, the former obtains. In any event, the author has reasons for the gap, but no excuses.

In Part I (in issue number 63, June, 1978), we observed how the Julian and Gregorian calendars were evolved and how they are described. Further, we observed how Joseph Scaliger obtained the Julian period of 7980 years; how he described a novel idea of counting days to yield a Julian Day Number; and, furthermore, how he obtained the epoch, or singular event, beginning the Julian period of 1/1/4713 BC. (An epoch is commonly a period or an era of some extent in time, but, astronomically, epoch also refers to an instant in time when some period commenced.)

In addition, we noted how the forever irritating year numbering error when crossing from 1 AD back to 1 BC came about. Now, we shall attempt to pull all these matters together, reconcile some descrepancies, and, hopefully, provide some rules, useful with computers, to calculate Julian Day Numbers from the date, and vice versa.

Toward the end of Part I, we noted a 60-day difference between the length, in days, of the Julian and Gregorian Julian periods of 7980 years. Since the two calendars differ by the leap year rules bis-a-vis century years, the two calendars differ in elapsed number of days by 3 every 400 years. Thus, since $7980/400 = 19 + 380/400$, there will be $19 \times 3 = 57$ less days in the Gregorian periods due to 76 centuries, plus 3 more days due to the 3 centuries in the 380 years left over, thus yielding 60 days difference.

The American Ephemeris and Nautical Almanac (AE&NA) published for astronomers, navigators, and astrophysicists every year, gives some tables for determining the Julian Day Number (JDN) of any given date from 1697 BC to 2296 AD, defined for suitable calendars. These tables are not trivial to use by any means and, furthermore, they perpetuate one source of one-day error which keeps cropping up in various calculations of JDN, to wit: the tables yield JDN's for day zero of any given month! This concept of a day 0 for January, for example, really means the last day of the previous month, or December 31 of the previous year. Careless preparers of programs to compute JDN often overlook this detail when testing their programs.

There is, of course, another source of persistant error of one day in the JDN; namely, the time of day. Since astronomers work at night, the time from dusk to the next dawn should all be a single day by their reckoning, while the civil calendar persists in changing its date at midnight for any given local time. This intolerable (to the astronomers) state of affairs is easily taken care of by defining the JDN to apply from noon at Greenwich; thus, in the Western world, encompassing all of any one night's observations into the same JDN. Thus, one must specify the date, the calendar used, and the time, if one wants an accurate determination of the JDN.

The AE&NA also gives a table near the front of the book called "CALENDAR, 19XX," and in that table, the "Julian Date" is given for each day of the year. This is different from the "Julian Day Number," since what is implied is both a date and a time of day. For example, January 1, 1979 has a Julian Date (JD) of 2443874.5 (that is, expresses the elapsed integral number of days from the Julian Period epoch, plus the fraction of a day since the last JDN incremented by one, and, refers to 0000 hours, Coordinated Universal Time at the Greenwich meridian.) Note that this JD correlates with the JDN given in the JDN tables (toward the back of the AE&NA); namely, January 0, 1979 (at noon)(really, December 31, 1978) has a JDN of 2443874 expressed as an integer. This fractional version of what should be an integer is another source of confusion about JDN's.

As another, but probably not final, source of confusion regarding JDN, the scientists responsible for certain aspects of the International Geophysical Year decreed that there would be a new epoch for what would be known as the Modified Julian Day (MJD) and that this would be 00 hours, Universal Time, November 17, 1858 AD, Gregorian Calendar. At that moment, the classical astronomer's JDN (really, JD) was 2,400,000.5! This introduction of a change in the JDN to a new MJD with a further adjustment of one-half a day seems to me to be totally unjustified and a typical example of bureaucratic nonsense. Alas, such is the way the world works.

The reader will recall that we examined a chart showing that 1 BC was the name of the year immediately preceeding 1 AD, thus depriving us of a year zero. This designation also causes some confusion for those who would calculate JDN from the date, and this is totally independent of the controversy regarding the year of the birth of Christ.

As a further minor point of confusion regarding calculations of JDN by many proposed schemes, numerous authors and texts fail to point out that integer arithmetic is essential when computing JDN or the date, and, furthermore, most (but NOT all) computer languages and their implementations perform non-frational residue arithmetic when dealing with numbers expressed as integers. For example, when one looks at an ALGOL procedure showing "year/4", where "year" is defined as integer, then "year" = 4, 5, 6, or 7 will all yield one for that quotient, and this is almost never described as the "greatest integer" function.

Now, finally, we have previously noted that not all of the world adopted the Gregorian calendar at the same time; indeed, some have not adopted it to this day! During the 16th, 17th, and 18th centuries, dates were frequently noted as being OS or NS (Old Style, that is, Julian Calendar; or New Style, that is, Gregorian). If these designations appear, they thus determine exactly which calendar is meant. Otherwise, for any given date, one must know which calendar is intended. We shall explore this issue a bit.

At the time Joseph Scaliger proposed the Julian Day Number scheme, he only knew the Julian Calendar, as we have previously stated; hence, all real dates prior to October 4, 1582 are with respect to the Julian Calendar or some calendar other than the Gregorian. If there is any reference to a Gregorian date earlier than October 15, 1582, it is an artificiality which exists only to extrapolate the current calendar backwards.

We shall make some arbitrary distinctions about dates for purposes of this discussion and for describing some computational schemes. Further, throughout the rest of this paper, unless a time-of-day is otherwise specified, we shall adopt the astronomer's definition of JDN; namely, that the number applies to a date as of noon Greenwich time until the next day at noon.

We shall conclude our discussion of sources of error in understanding the JDN by exposing one final egregious selection by Joseph Scaliger. Like the scientist that he was, he chose to begin his numbering of serial numbered days with zero for the first day and not one. This allows one to use the JDN as both a serial number (hence, name) for a given date, and also as a count of elapsed days up to the beginning of the given astronomical day.

Scaliger's cleverness has been the source of endless confusion about JDNs for four centuries. Isn't it marvelous how we humans can complicate our lives? Perhaps this will all be somewhat clearer if we examine a chart which diagrammatically correlates all the items we have mentioned.

We shall conclude our discussion of sources of error in understanding the JDN by exposing one final egregious selection by Joseph Scaliger. Like the scientist that he was, he chose to begin his numbering of serial numbered days with zero for the first day and not one. This allows one to use the JDN as both a serial number (hence, name) for a given date, and also as a count of elapsed days up to the beginning of the given astronomical day. Scaliger's cleverness has been the source of endless confusion about JDNs four four centuries. Isn't it marvelous how we humans can complicate our lives? Perhaps this will all be somewhat clearer if we examine a chart which diagrammatically correlates all the items we have mentioned.

Let us look at Figure 1. Row I spans about three days of elapsed time with some points marked for reference. We are, of course, looking at events from the meridian of Greenwich, and are referring to Universal Time as defined for that line.

Row II gives the Eccliesiastical or Civil date in the Gregorian calendar as it is most commonly accepted in the Western world. The day of the week is also given, since this is uniquely determined by the use of specific dates.

Row III gives the astronomical day as used prior to 1925. This form corresponds to the Julian Day Number period of Noon to the following Noon; its use was generally discontinued after 1925.

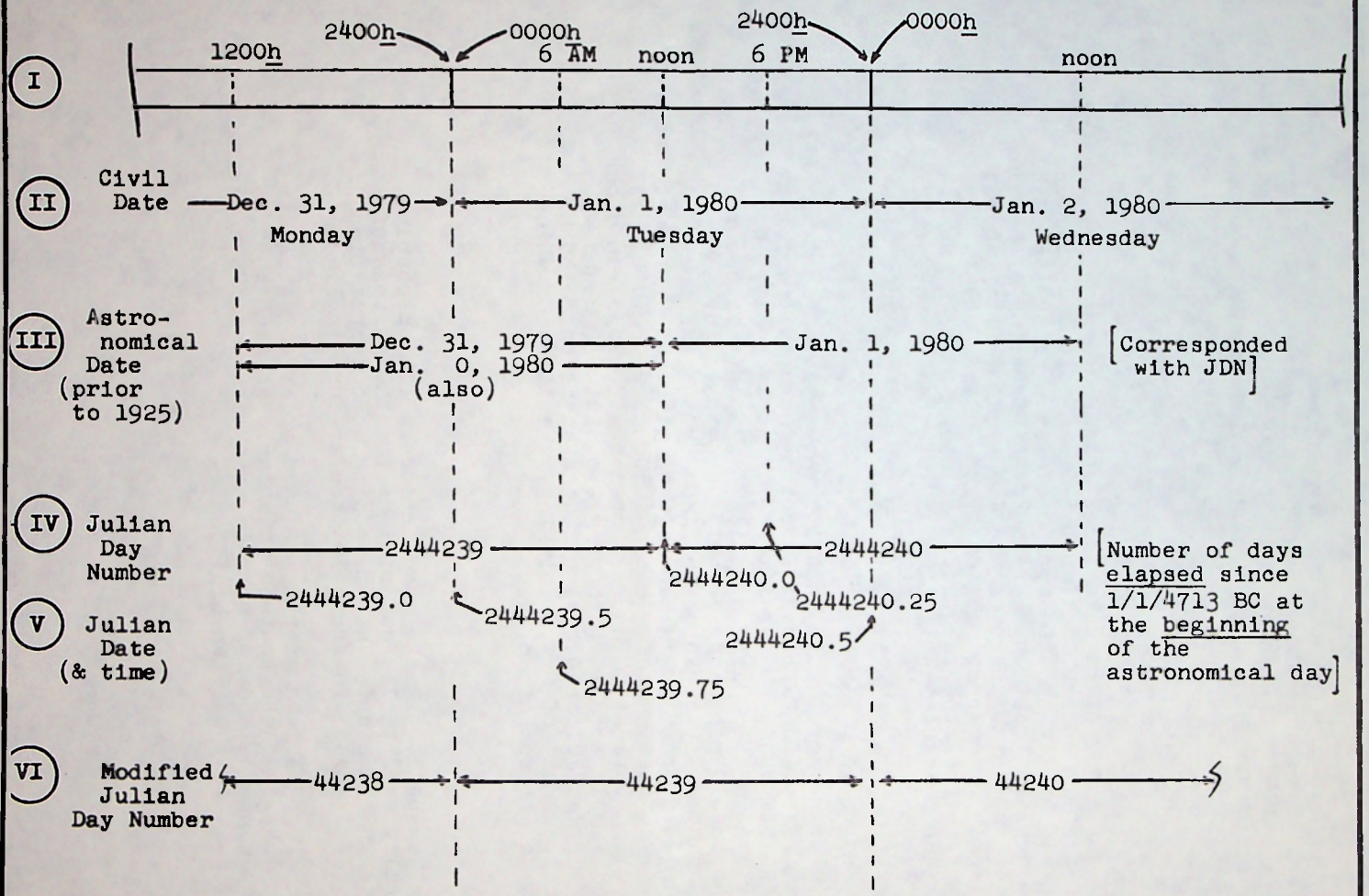

I
1200h
2400h
0000h
6 AM
noon
2400h
6 PM
0000h
noon
II
Civil Date
Dec. 31, 1979
Monday
Jan. 1, 1980
Tuesday
Jan. 2, 1980
Wednesday
III
Astro-nomical Date (prior to 1925)
Dec. 31, 1979
Jan. 0, 1980 (also)
Jan. 1, 1980
[Corresponded with JDN]
IV
Julian Day Number
2444239
2444240
2444240.0
2444240.25
[Number of days elapsed since 1/1/4713 BC at the beginning of the astronomical day]
V
Julian Date (& time)
2444239.0
2444239.5
2444239.75
2444240.5
VI
Modified Julian Day Number
44238
44239
44240

Figure 1

Row IV gives the Julian Day Number for each illustrated day and shows the extent of its application. This JDN is the serial number name of that day and also the number of elapsed days since the Julian Period epoch of January 1, 4713 BC, as measured at the start of the named astronomical day, up to noon of that day.

Row V begins to make it all clear, we hope, by showing what is known as the "Julian Date," a decimal number whose integral part is the JDN applicable at a given time, and whose decimal fraction represents the fraction of the day elapsed since the start of the JDN. Note that when the civil date changes at midnight, the Julian Date is ...XX.5, expressing the fact that one-half of a day has elapsed since the last Julian day started (at noon the previous day).

Furthermore, note that if one carelessly understands JDN, that there are two different numbers, differing by one, which could be associated with a given civil date. Row VI shows the Modified Julian Day Number as adopted during the IGY and subsequently perpetuated by some accountants. It can be obtained by subtracting 2,400,000.5 from the Julian Date at midnight of the start of a given civil date. This value of 2,400,000.5 is the Julian Date at 00 hours, November 17, 1858; an epoch picked simply to allow dropping two digits (2 and 4) from current Julian Dates! This seems to me to be a typical example of fuzzy thinking, the likes of which crops up all too often, alas.

We shall now make some arbitrary (but probably generally acceptable) definitions for the purpose of expressing some dates and proposing some computational schemes. These are shown in tabular form in Figure 2.

Figure 3 is a listing of some specific dates with their JDNs. Note that in this table we give the JDN for the astronomical day of noon-to-noon with the JDN applying after noon of the given civil date.

Dates	Applying to
prior to 1/1/4713 BC	Do not apply to JDN calculations (of interest at this time)
1/1/4713 BC to 10/4/1582 AD	Julian calendar
10/5/1582 to 10/14/1582	Dates which can only exist in the Julian calendar, but which coincide exactly with 10/15/1582 to 10/19/1582 of the Gregorian calendar.
10/15/1582 to 12/31/3267	Gregorian calendar
1/1/3268	Do not apply to JDN calculations (of interest at this time)

Further:--

- If a date has BC or AD appended, it is taken as that, but in the absence of these, it is taken as AD.
- If a date is marked OS or NS, it is taken to mean Julian or Gregorian respectively, unless it violates the Gregorian beginning rule of 10/15/1582. (OS dates can persist clear up to 12/31/3267 AD.)
- If a date is written with a - (minus) before the year, it is taken to be BC.
- Only dates between 1/1/4713 BC and 12/31/3267 AD are valid.

Figure 2

Some Specific Dates

Date	JDN	Remarks
1/1/4713 BC	0	Scalawag Scaliger did this to us!
1/1/4712 BC	366	Yes, 4713 BC was a leap year.
1/1/1 BC	1721058	One BC was also a leap year (see Figure 4)
12/31/1 BC	1721424	Note the absence of a year zero.
1/1/1 AD	1721425	
10/4/1582	2299160	Pope Gregory's date
10/15/1582	2299161	Gregorian Calendar
10/15/1582 OS	2299171	Julian Calendar
1/1/1584	2299604	Note (1) below
1/1/1900	2415021	Note (2) below
1/1/1979	2443875	
1/1/1980	2444240	
5/1/1980	2444361	A proleptic date, perhaps

Note (1). This date is given because it can be checked in the American Ephemeris and Nautical Almanac and can be computed from 10/15/1582 as follows: to 11/1/1582, +17; to 1/1/1583, +61; to 1/1/1584, +365; or, from 10/15/1582 to 1/1/1584, add 443.

Note (2). Although we call this the start of the "nineteen hundreds," it is not truly the start of the 20th century; 1/1/1901 starts the 20th century because of our old friend Dionysius.

Figure 3

Figure 4 shows the numbering of the years as the BC/AD boundary is crossed, and, further, shows the occurrence of leap years in the BC era back to the beginning of the Julian Period, 4713 BC. Since these dates refer to the Julian calendar, there are exactly 25 leap years each century plus the four shown for the period 4713 BC to 4701 BC. Thus, 47 (centuries) x 36525 (days per century) + 9 x 365 (non-leap years between 4713 and 4701 BC) + 4 x 366 (leap years) = 1721424 = number of days elapsed in the Julian Period from 1/1/4713 BC through 12/31/1 BC. In other words, this is the number of BC days there were in the first Julian Period. Referring again to Figure 1, we can see that commencing at noon, 1/1/1 AD, the JDN of the first day of the Christian Era is 1721425 (as listed in Figure 3). (Notice that we have carefully avoided ending a sentence with a JDN so that the period at the end of a sentence would not be confused with a decimal point--these appear only in Julian Dates.)

We are promised a Part III of "Measuring Time" in which Mr. Parkin will discourse on the actual calculation of JDNs.

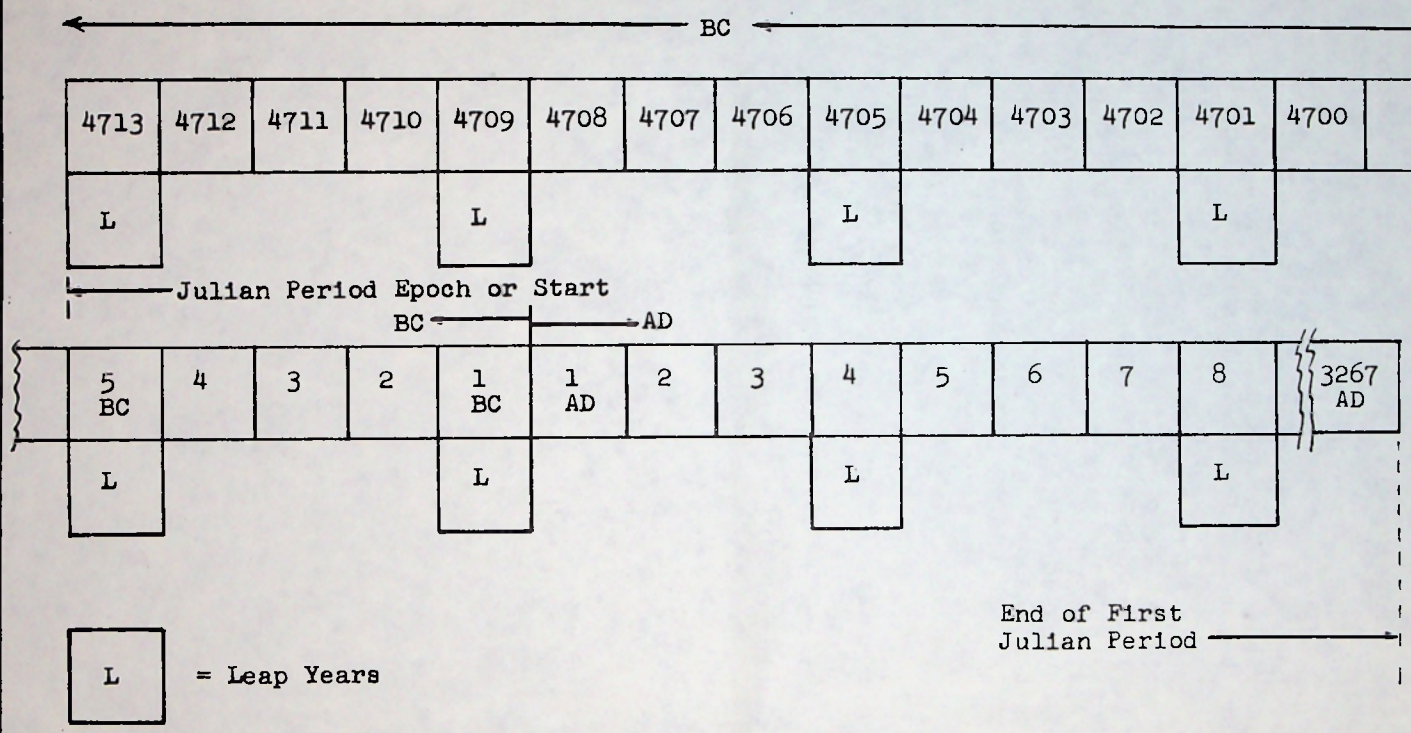

BC
4713
4712
4711
4710
4709
4708
4707
4706
4705
4704
4703
4702
4701
4700
L
L
L
L
Julian Period Epoch or Start
BC
AD
5 BC
4
3
2
1 BC
1 AD
2
3
4
5
6
7
8
3267 AD
L
L
L
L
End of First Julian Period
L
= Leap Years

Figure 4

A Star Is Formed

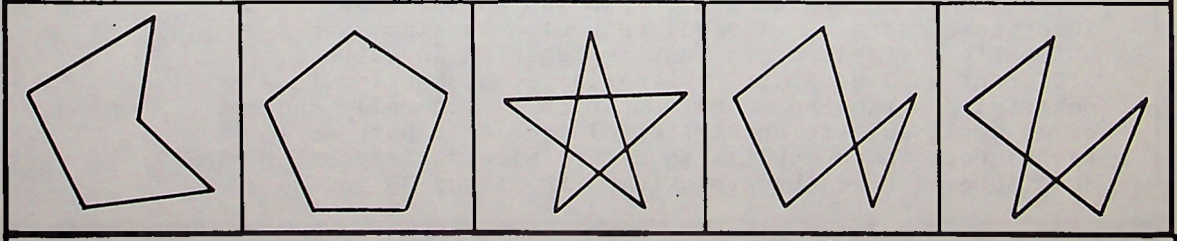
by Thomas R. Parkin

(Including a few words on recursion)

In issue 36, Problem 119, titled "The Bicentennial Star," illustrated several 5-point connected figures, one of which is a 5-pointed star such as we have all learned to draw in grammar school. Given five randomly located points in a plane, connecting them in the order of their being chosen will not, in general, yield a typical 5-pointed star, independent even of scale distortions. To answer the first problem posed by #119, it is probably necessary to perform a computer simulation experiment with various parameters and study the results. However, the second part of the question; namely, "What is the logic of distinguishing the star from other shapes?" is something we can fairly easily answer. The algorithm for this process can be heuristically described as follows:

Number the points in the order in which they are selected. Consider the line joining any pair of points: i and i+1. Then, points i+2 and i+3 must lie on opposite sides of that line AND points i+3 and i+4 must also lie on opposite sides of that line; where (i+j) is reduced modulo 5. [Another way of saying this is to note that i+3 must lie on the opposite side of that line from i+2 and i+4 (which must lie on the same side of that line). Repeat this test for each pair of points in turn, and only if the test is passed for each pair, is the set of 5 points, when connected in the order numbered, a 5-pointed star.]

Now, of course, there are some practical details for implementing this algorithm, and we shall consider some of them. For example, testing a given set of 5 points can halt as soon as the test fails once, since no star can result. Further, in the subroutine which tests for opposite-sidedness, some limit of closeness to the line must be chosen, corresponding to the precision of one's calculations. Similarly, in the random selection of points for consideration, some total area must be set as the arena of exploration. Assuming that these practical details can be settled, the problem, then, is solved.



We shall digress for a bit to explore some words and concepts. The solution of a problem by the use of computers is basically a two-step process, exclusive of the programming and running on the machine. These steps are: (1) What to do, and (2) How to do it.

"What to do" is answered usually by the algorithm we choose. An algorithm generally implies a set of directions for solving a problem; it is not an implementation of a problem solution. An algorithm, in a formal mathematical sense, can be precisely described and has certain properties of exactness, determinacy, generality and halting. These formal ideas are usually not implied when we use the term "algorithm" to describe a scheme for solving some particular problem, although in a vague sense they are subsumed.

We shall not dwell on mathematical rigor regarding our use of the term "algorithm," but we will note that for our purposes of solving problems by computers, an algorithm is our road map of "what to do," and not a complete set of directions for "how to do it." Furthermore, an algorithm may be short and very simple, or quite complicated and long. Likewise, it may be very easy to implement or it may be innocent sounding and almost impossible of execution in practical terms.

The "how to do it" part of our problem solution frequently involves selection of a method of implementation for our algorithm; here is where we begin to hear words like backtracking, loops, recursion, subroutines, and a myriad of other named programming techniques. Indeed, that is exactly what these words are; namely, names for programming techniques which can be applied variously in the specific implementation of an algorithm. For instance, we give the name "subroutine" to a group of instructions we wish to execute repetitively, and this concept is also known by the names "function" and "procedure," among others. Indeed, we have many synonyms in the computer field, since clever ideas keep recurring to new people who simply haven't learned the old words for them!

As another example, we should note that the technique of recursion can be applied almost anywhere a loop can be used. Indeed, recursion is simply an automatic variable depth loop implemented in quite a different way, usually. Likewise, backtracking is a powerful programming technique which is generally thought of as applicable to combinatorial problems, but, as a technique, can be applied to a very wide variety of problem solutions. (See "Backtracking," in issues 33 and 34.)

But let us return for a moment to recursion. The concept of recursion is usually illustrated by the calculation of the factorial:

$$n!, \text{ meaning } n\cdot(n-1)\cdot(n-2)\cdot(n-3)\cdots(2)\cdot(1).$$

It stems from the obvious relation that factorial n is equal to n times factorial (n-1), or, as an equation:

$$n! = n\cdot(n-1)!$$

Thus, we have an expression for something in terms of itself in a compact mathematical way. But this does not preclude our using the idea of recursion for other than tidy mathematical expressions. We shall illustrate this concept by showing how to solve the original STAR problem by several implementations, including recursion, all of them programming the same algorithm previously described.

The first thing we will do is assume a subroutine, which, when given the indices of two consecutive points (or, at least one such index), can test the opposite-sidedness of the appropriate pairs of points, and return a "pass" or "fail" signal for our further edification. We shall devise this subroutine later. Furthermore, we shall devise our data storage mechanism later as well. At this point we shall concentrate on the "how to do it" part, first using a loop and secondly using recursion.

Since this article is fundamentally expository, and not a specific program example, we shall use a pseudo-code language which we shall not define, but which should be more or less obvious to anyone familiar with BASIC or Fortran or ALGOL. Furthermore, we shall use the same gross flowchart for each implementation--Figure 1. The variation will occur in the third box, indicated by arrows, and we shall not consider the details of the remainder of the flowchart.

Now for the loop approach. We shall want to vary our loop index over a range of 5, and we shall need some other indicies depending on that loop index. Since none of our indicies ever exceeds 5, we shall use the modulo mechanism and allow our loop index to run from 0 to 4 and then, as we form indicies from the loop index by addition, we can reduce those formed indicies modulo 5, and simply use the residues or remainders. (If <u>your</u> language does not allow a zero index for a list or a loop, then <u>you</u> must accommodate the problem!) Next, let us look at the general case again--the OPPSIDE subroutine we shall call needs to be given a parameter and then it can form the index of each point it needs to consider from that parameter. For instance, suppose we give the subroutine the loop index, i, as its parameter. Then OPPSIDE will:

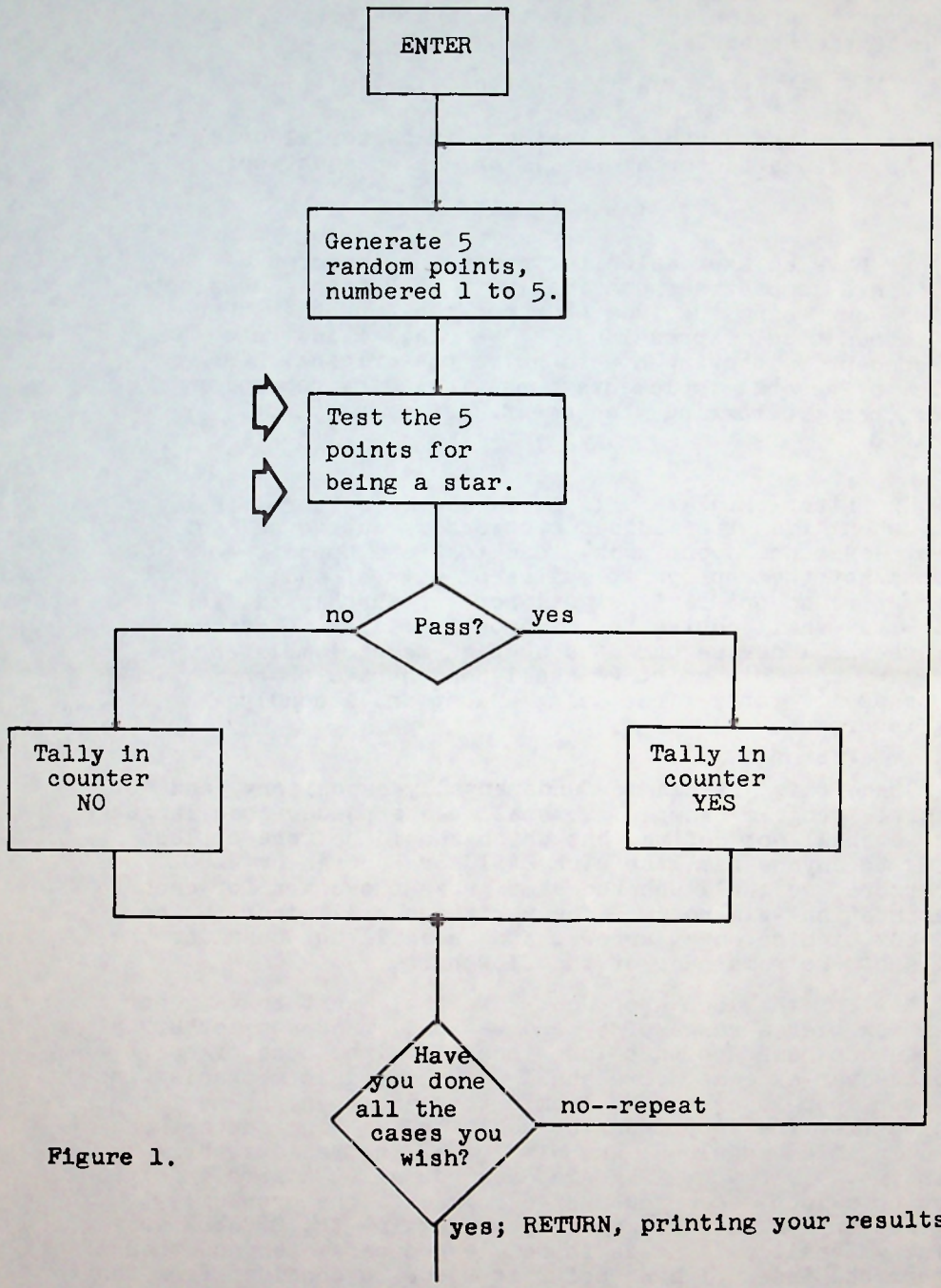

ENTER
Generate 5 random points, numbered 1 to 5.
Test the 5 points for being a star.
Pass?
no
yes
Tally in counter NO
Tally in counter YES
Have you done all the cases you wish?
no--repeat
yes; RETURN, printing your results

Figure 1.

1) Form the line from point i to point (i+1)

2) Check that points (i+2) and (i+3) are on opposite sides of that line

3) Check that points (i+3) and (i+4) are on opposite sides of that line, and

4) If both checks pass, return OK to the calling routine.

Of course, OPPSIDE will know how to avoid getting point numbers other than points 1 through 5 (or 0 through 4, as the case may be), and OPPSIDE will know how to perform the actions attributed to it. We shall see how, later.

For now, we can test the exit flag from the sub-routine OPPSIDE each time we call it from inside our loop. And, finally, (we are almost done), we will set up our loop exit result of PASS or FAIL for our main program to examine. We shall be clever enough to avoid executing the loop more than enough times to either PASS five times or FAIL once, and, thus, we shall really go a variable number of times around the loop--but that has not really changed the loop approach. Figure 2 is our loop in our version of pseudo-code.

Now, before we look at the recursive way to implement this same loop, we must first detail out our subroutine OPPSIDE.

```
RESULT ←— PASS
FØR i = 0 TØ 4
      CALL ØPPSIDE(i,f)
      IF f = ØK
            THEN NEXT i
            ELSE RESULT ←— FAIL
      ENDIF
ENDFØR
```

Figure 2.

Subroutine OPPSIDE (short for "test for opposite-sidedness") does the four steps previously listed. Of course--it sounds easy--but, exactly how? Let us fall back on some simple coordinate geometry, or what has become known as analytic geometry. We recall that in the x-y plane, with rectangular coordinates, we can show that every straight line has a simple linear equation of the form $Ax + By + C = 0$. Furthermore, if the line passes through the two points whose coordinates are (x_1,y_1) and (x_2,y_2), then the equation of that line can be written:

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1}$$ (the 2-point form)

Some simple algebra shows us that:

$$A = y_1 - y_2$$

$$B = x_2 - x_1$$

$$C = x_1y_2 - x_2y_1$$

Thus we have transformed the coordinates of two points into the standard form of the equation of a line. The next thing we need to recall is how to compute the distance from a given point to a given line. The formula is this: the perpendicular distance of the point with coordinates (x_3,y_3) from the line $Ax + By + c = 0$ is:

$$\frac{A \cdot x_3 + B \cdot y_3 + C}{\pm\sqrt{A^2 + B^2}}$$

where the sign before the radical is opposite to that in C when $C \neq 0$, but the same as that in B when $C = 0$.

Now we have all the tools we need. Of course we should remember that the sign of the perpendicular distance from a point to a line is different depending on which side of the line it lies on--exactly which direction is + and which is - is quite arbitrary and is usually chosen by convention to be + for "upward" or "to the right," but suffice to say, when two points lie on opposite sides of a given line, then perpendicular distances will have different signs.

Well, isn't that just dandy? All we need to do now is some simple arithmetic, in a carefully organized way, and we have devised the subroutine OPPSIDE.

At this juncture we shall pause to organize our data storage. The data we need to work with consists of the coordinates of five points, the values chosen at random in a particular order, generated by our problem driver program. We shall place these in a table which we can think of as looking like this:

Serial number of point	Index	x coordinate	y coordinate
1	0	x_0	y_0
2	1	x_1	y_1
3	2	x_2	y_2
4	3	x_3	y_3
5	4	x_4	y_4

[A diversion: In programming, one frequently finds a situation where some space can be traded for time--and here is such a case. If the table data is simply repeated in order in index slots 5 through 9, then all table look up actions referring to, say, i+5, when i = 4, then simply refer to index 9 rather than having to be reduced modulo 5 to slot 4. This saves time for each index calculation.]

We are now in a position to write the pseudo-code for our subroutine to implement the four steps. Of course, we do not have to explicitly determine the equations for various lines, but simply recognize that given the index of one point, the coordinates of that point and the next one in order determine a line, the coefficients of which equation we can easily calculate.

Let us look at an example, Figure 3. Using the five points, P(1) through P(5), as listed in the upper part of the table, we can proceed to the calculation of A, B, and C for each pair of consecutive points as shown by i = 0 and i = 1 (we leave i = 2, 3, and 4 as an exercise for the reader). (We shall examine P(4') later.) Variables q, r, and s are simply names for the perpendiculars from points (i+2), (i+3), and (i+4) to the line formed from points i and (i+1). The equations are:

$$q = A \cdot x_{i+2} + B \cdot y_{i+2} + C$$

$$r = A \cdot x_{i+3} + B \cdot y_{i+3} + C$$

$$s = A \cdot x_{i+4} + B \cdot y_{i+4} + C$$

The test to establish the OK or NG status is to verify that q and s have the same sign and that this is opposite to the sign of r. If this obtains, return OK; otherwise, return NG. See Figure 3A.

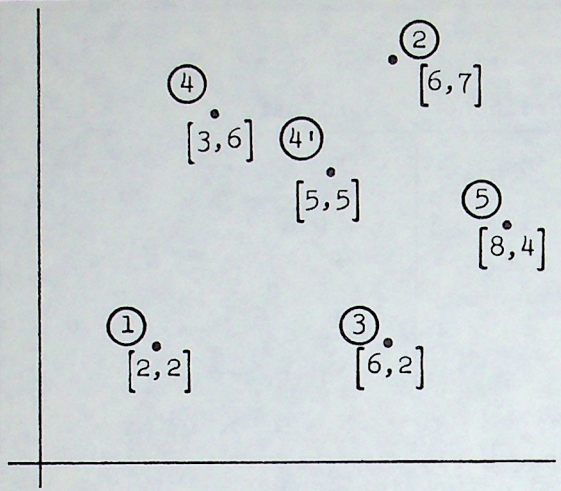


Table of Points

P	i	x_i	y_i
1	0	2	2
2	1	6	7
3	2	6	2
4	3	3	6
5	4	8	4
4'	3'	5	5

Figure 3

i	A	B	C	q	r	s	Status
0	-5	4	2	-20	+11	-22	OK
1	5	0	-30	-15	+10	-20	OK
0'	-5	4	2	-20	- 3	-22	NG

(Using i = 3' for P(4'))

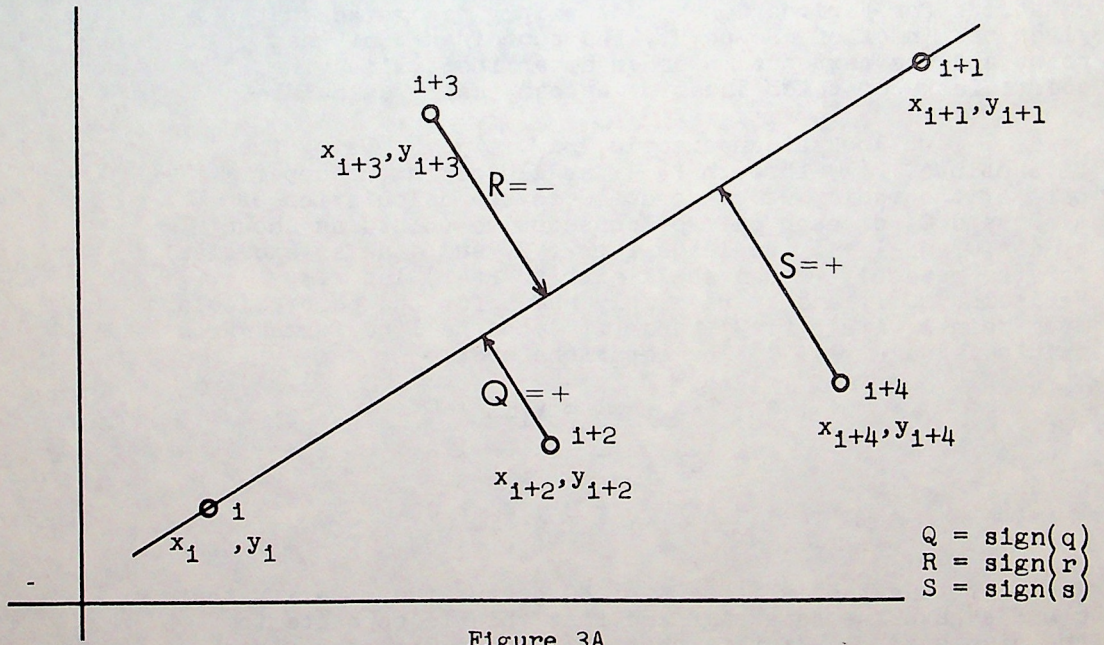


Figure 3A

Now the careful reader will notice that we did not divide by

$$\pm\sqrt{A^2 + B^2} \qquad \text{(call this W)}$$

and we did not choose the sign of the radical by the rules we gave. Further reflection will show that since A, B, and C are the same for any one set of q, r, and s, then dividing by some constant (such as W), or making an arbitrary but uniform sign selection, will make no difference to the validity of the result. The size of the numbers might change, and the signs might all be reversed, but these phenomena will have no effect on the logic of the result. However, if one wants to know the magnitude and direction of the perpendiculars, in the same terms as the scale of the coordinates of the points, then one must select the proper sign of the radical and compute the value of the radical to use as a divisor.

The pseudo-code for the subroutine OPPSIDE is shown in Figure 4, where we have introduced two additional notions. First, we use the SGN function which returns +1, 0, or -1 depending upon the sign of the expression, with the zero corresponding to a zero expression value. Secondly, we introduced tests for these zero values which would, of course, indicate collinearity of three (or more) points.

```
ENTER
f ⟵ NG
A ⟵ y(i) - y(i+1)
B ⟵ x(i+1) - x(i)
C ⟵ x(i)·y(i+1) - x(i+1)·y(i)
Q ⟵ SGN(A·x(i+2) + B·y(i+2) + C)
IF Q = 0 THEN RETURN
R ⟵ SGN(A·x(i+3) + B·y(i+3) + C)
IF R = 0 THEN RETURN
S ⟵ SGN(A·x(i+4) + B·y(i+4) + C)
IF S = 0 THEN RETURN
IF Q ≠ R THEN

      IF Q = S THEN
            f ⟵ ØK

      ENDIF

ENDIF
RETURN
```

Figure 4.

Subroutine OPPSIDE(i,f)

Let us go back to Figure 3 and look at the point P(4') with coordinates (5,5). If we substitute this point for point P(4) with coordinates (3,6) and repeat the call to our subroutine with i = 0, then the numbers shown in the line for i = 0' will result, and the status will be NG because the signs of all three, q, r, and s, are alike. The reader can connect the various points, in order, to see that P(1) through P(5) do constitute a five-pointed star, while using P(4') in place of P(4), degenerates the figure so that it is not a five-pointed star.

So much for the loop approach to implementation. We shall now contemplate recursion. Recursion is an implementation technique in which subroutines or functions are able to invoke themselves to any useful depth without the program getting lost. In most early (and many current) programming languages, a subroutine, when called in a program, explicitly stored in itself, or in some uniform, fixed, single address location, the address from whence it was called. Thus, when the subroutine was ready to return to the calling location, it knew where to go. However, if the subroutine were allowed to call itself, most systems would then lose the location of the original caller and the program would not run correctly. In machines, or under software systems, where a "stack" is maintained, then repeated calling of a subroutine by itself could be allowed, since the return address for each call to the subroutine could simply be stored on the stack and be retrieved when needed. Thus it is said that "stack machines support recursion"--but this is also true for some languages which run on non-stack machines, such as ALGOL on the CDC 6600, because in those cases, either the run-time environment of the language implementation or the operating system under which it is run, support a software simulated stack. Such software implementations of stacks are sometimes a bit slow, and hence are not too popular with users.

A recursive subroutine (or function, but since these are not different in essential substance, we shall refer only to subroutines) must be able to stop calling itself or we should be faced with the endless effect of two facing mirrors--no end in sight!

In the typical example used to illustrate recursiveness, the factorial function, the subroutine usually has a test at its beginning for n = 1 or n = 0 or both. In the case of n = 0, n! = 1 by definition, and that is returned. In all other cases, when n = 1, either in the initial call to the subroutine or at any subsequent call when the passed parameter has been reduced to 1, then the subroutine returns a value of 1 and exits.

If the exit is to a stacked series of calls, the subroutine will perform its action repeatedly and work its way through the stacked calls until it gets to the ultimate original caller, whereupon the problem is done. For an important, generally useful subroutine such as FACTORIAL, the implementation will usually have tests for the proper range and form of the parameter passed to it to preclude a program malfunction--but that is the topic of another paper.

Note that the subroutine which is recursive does not know how many times it will execute, and, in this sense, is somehow equivalent to a loop with a variable upper limit to its control index. Indeed, we could say that the recursive subroutine is functionally identical to a loop with a variable index and we shall show how that appears by our STAR example.

The subroutine OPPSIDE is now modified so that it can call itself, know when to stop, and when to return; this will make our subroutine into the implementation of the entire marked box of Figure 1. Figure 5 is a flowchart of this new version and Figure 6 is the pseudo-code.

```
ENTER
f ← NG
A ← y(i) - y(i+1)
B ← x(i+1) - x(i)
C ← x(i)·y(i+1) - x(i+1)·y(i)

Q ← SGN(A·x(i+2) + B·y(i+2) + C)
IF Q = 0 THEN RETURN f

R ← SGN(A·x(i+3) + B·y(i+3) + C)
IF R = 0 THEN RETURN f

S ← SGN(A·x(i+4) + B·y(i+4) + C)
IF S = 0 THEN RETURN f

IF Q ≠ R THEN
      IF Q = S THEN
            IF i < 6 THEN
                  CALL S/R ØPPSIDE(i+1,f)ELSE f = 1
            ENDIF
ENDIF
RETURN f
END S/R
```

Figure 6.
Subroutine OPPSIDE(i,f)
[Recursive version]

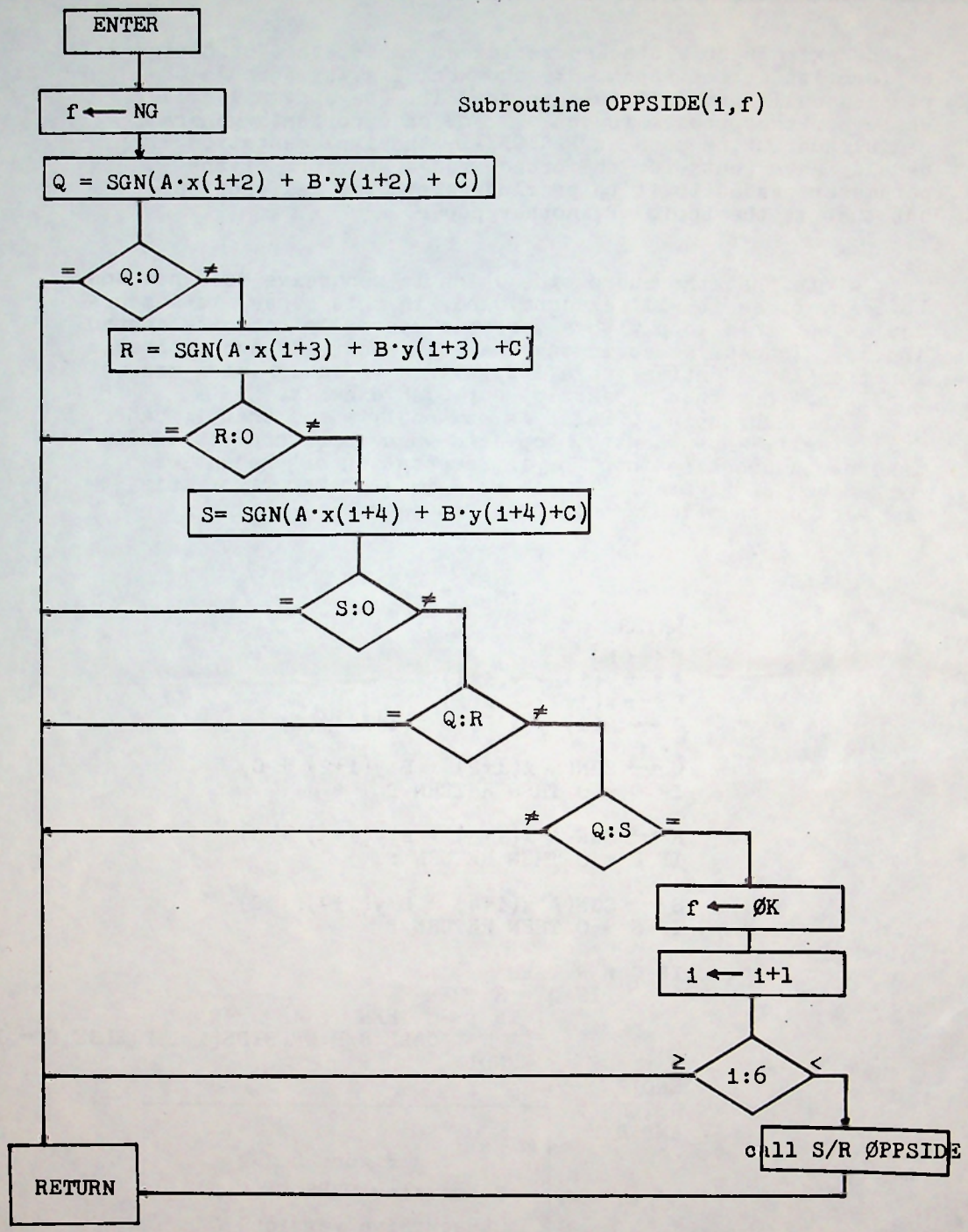

ENTER
f ← NG
Subroutine OPPSIDE(i,f)
Q = SGN(A·x(i+2) + B·y(i+2) + C)
=
Q:0
≠
R = SGN(A·x(i+3) + B·y(i+3) +C)
=
R:0
≠
S= SGN(A·x(i+4) + B·y(i+4)+C)
=
S:0
≠
=
Q:R
≠
≠
Q:S
=
f ← ØK
i ← i+1
≥
i:6
<
call S/R ØPPSIDE
RETURN

Figure 5